Note: I'll be updating this for the 1.10 Edition Kelsey put out, in the near future.
https://github.com/kelseyhightower/kubernetes-the-hard-way/tree/1.10.2

Introduction:

The goal of this exercise is to take Kelsey Hightower's "Kubernetes The Hard Way" (https://github.com/kelseyhightower/kubernetes-the-hard-way/tree/1.7.4, using the 1.7.4 version) and provide an adaptation that is based on local VMs, without the use of cloud infrastructure, specifically avoiding the GCloud commands in favor of keeping your IPs visible, and doing some more manual typing. I think Kelsey's work is excellent, but is almost TOO streamlined. It WILL work as written, which minimized my opportunity to learn from mistakes made.

The purpose of this exercise is really to make those mistakes, and highlight them along the way. Cut-and-paste will shoot you in the foot due to the number of IPs and names that have changed, so you WILL be forced to read all the configs, yaml, etc…

WARNING 1: This document is very "stream of consciousness" and is intentionally not holding your hand. If you're reading this doc, you're here to learn, so you need to really be aware of what you're doing.

WARNING 2: This document is not all-inclusive. You will need to have Kelsey's work open, and use this document as an adaptation/study guide.

In the near future, I'll do a comparison of using v1.7.4, as this doc was written for, against the newer editions, probably starting with ~~v1.9~~ v1.10

The following video is a presentation of the unfinished product to the CKA Office Hours:
https://youtu.be/K6gK7euL8Fo

There's a couple major takeaways comparing the above video to the finished state:

1) I apparently had no idea what kube-proxy was supposed to be doing. I thought it was an interface between the kubelet and the api-server, not for the app-layer. Whoops.
2) I was really spinning my wheels until I found the solution to the kubelet issue. It was a serious case of grasping at straws, and just knowing where to look was still miles away.
3) I say "Uh" and "Ummm" a lot early in the morning.

Okay, on to the actual Doc

Prerequisites (1)

I'm using Virtual Box (v5.1) instead of GCloud. This means many of the pre-defined bash steps will become manual work. I did this to get a better understanding of all commands I was using, and also to commit things to muscle memory. In any case, your preq's are:

Local: VBox, and ideally an ssh client
Servers: bash, curl, wget

Yeah, not an extensive list.

Installing the Client Tools (2)

This section did not change, however I used my "kubemaster" VM for the client tools installation, so I did this step after provisioning the compute resources (the next exercise). If you want to use your local laptop, or another VM, that's fine, as long as they can communicate via the networking defined in the next section.

Provisioning Compute Resources (3)

It's important to note that, for the purposes of this exercise, since all VMs are on a single piece of hardware, HA really doesn't exist, so I'm skipping the steps that involve multiples of a single node type (controllers, workers, etc).

First, I built the 3 nodes:
kubemaster - 192.168.2.100
kubenode - 192.168.2.101
ketcd - 192.168.2.102

All VMs are 1 CPU, 2GB, and ~20GB of dynamic disk (default layout, no LVM). I disabled Floppy and Audio, as they were irrelevant. All other non-network options stayed default, with networking on the Host-only adapter. I installed the nodes with Ubuntu 16.04, choosing only the "base" and "ssh server" packages, then immediately ran apt-get update and apt-get upgrade.

In the initial incarnation, the networking was not portable, as the IPs my VMs had in my office were not the same IPs they would have in my house, for example. So, I ALSO built a VM using Untangle (https://www.untangle.com/get-untangle/) to act as a network bridge, and if I wanted to, I could also have easy firewalling and related apps for further testing and simulations. The front of the Untangle box is set as a bridged adapter to my laptop's NIC (and can be alternated between wired and wireless, as necessary) using DHCP, and the back NIC is 192.168.2.1 on the Host-only network with the other 3 main VMs. This solves my portability problem, as long as the Untangle VM is stood up and working before the others.

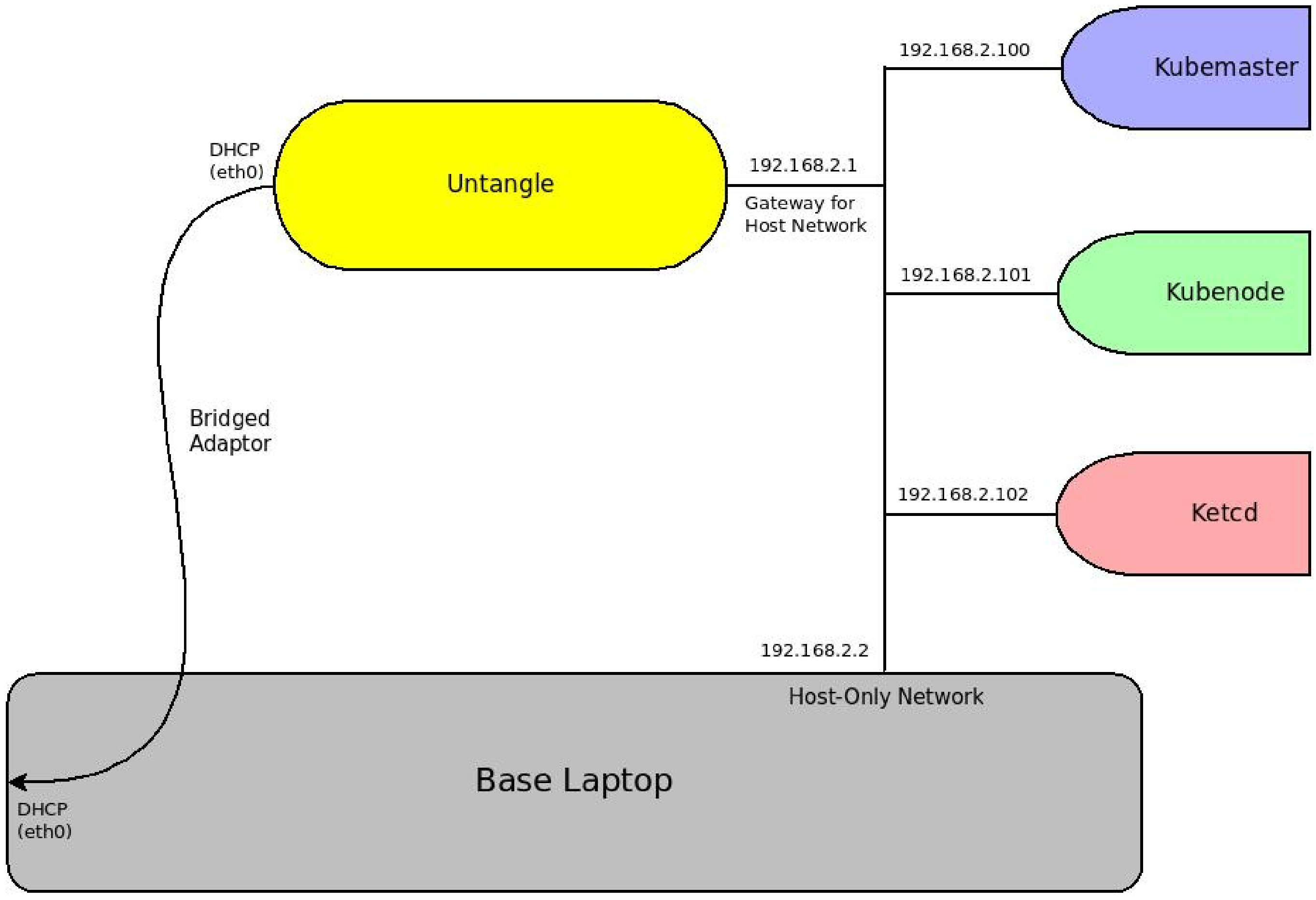
The Untangle VM was also 1CPU, 2GB, 20GB HD, but I gave it 32MB of Video memory, as it is primarily GUI-driven.

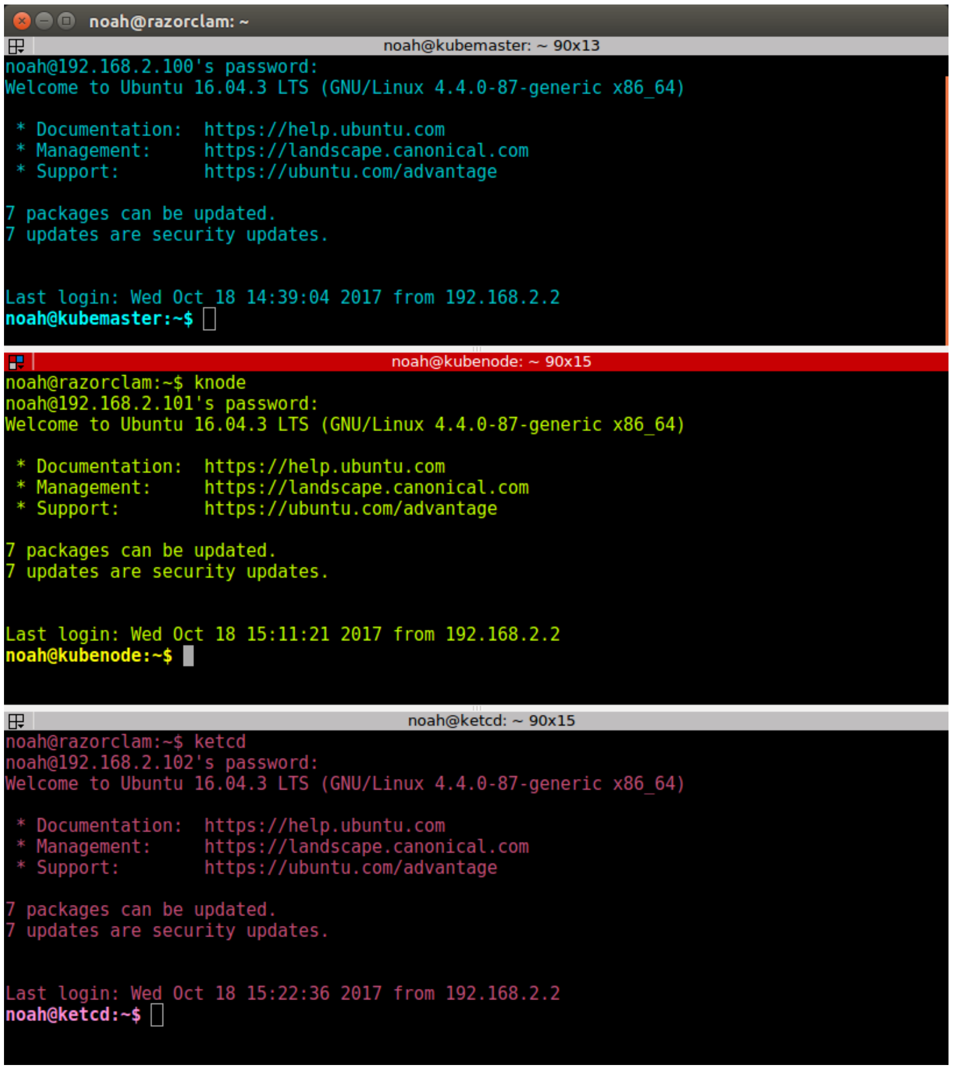
Now my /etc/hosts looks like this on each node:

```
127.0.0.1       localhost
192.168.2.100   kubemaster
192.168.2.101   kubenode
192.168.2.102   ketcd
```

Each of the nodes also points to the inside of the Untangle VM (192.168.2.1) at the primary nameserver. Untangle gets its DNS via the external DHCP. No other DNS configuration is used in this example.

Network Diagram:





```
noah@192.168.2.100's password:
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.4.0-87-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

7 packages can be updated.
7 updates are security updates.


Last login: Wed Oct 18 14:39:04 2017 from 192.168.2.2
noah@kubemaster:~$
```

```
noah@razorclam:~$ knode
noah@192.168.2.101's password:
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.4.0-87-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

7 packages can be updated.
7 updates are security updates.


Last login: Wed Oct 18 15:11:21 2017 from 192.168.2.2
noah@kubenode:~$
```

```
noah@razorclam:~$ ketcd
noah@192.168.2.102's password:
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.4.0-87-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

7 packages can be updated.
7 updates are security updates.


Last login: Wed Oct 18 15:22:36 2017 from 192.168.2.2
noah@ketcd:~$
```

^ Typical set of Login sessions

This also means we skip the whole “Networking” Section, because we don’t have a VPC, we don’t need the firewall (because the VMs are all Host-only), and we don’t have to differentiate between public and internal addresses.

CHALLENGE MODE: If you want this to be REALLY difficult, and therefore a great learning experience... Before you go on to the next step, follow the first few kubeadm instructions and install into both the controller and worker node, then try to undo what kubeadm did for the rest of the exercises.
NOTE: This is a bad idea, and you should use clean nodes. I’m only suggesting it to make your life harder.

Provisioning the CA and Generating TLS Certificates (4)

We skipped the steps which were specific to either gcloud (we already had the IP addresses) or multiple nodes (only need to generate certs for 1 node).

The 10.32.0.1 address is the first service in the cluster ("kubernetes" itself) if the cluster is assigned 10.32.0.0/24 as the service-cluster-ip-range (must not overlap with any pod IPs) in kube-apiserver service configuration.

Now, do we actually need to make this cert line up to include etcd? Or could we make our own etcd certificate? Because the api-server has cli flags of etcd-certfile and etcd-keyfile, I think making a separate key is ideal, but which key should sign it? Just the base CA key? Yeah, we're going to use a single CA to make life easier.

The hostname in the cfssl gencert (which will be in --tls-cert-file for the api server) needs to include a match to the "cluster: server: https://name:443" in the kubeconfig of the kubelet, kubectl, etc. Also, the kubeconfig has to explicitly define the path to the ca cert file, not just add it to /etc/ssl or use update-ca-certs or whatever.

If you decided to pass everything through the Untangle box, you would need to include the public IP of that in the -hostname list of the following:

```
cfssl gencert  -ca=ca.pem   -ca-key=ca-key.pem  -config=ca-config.json
-hostname=10.32.0.1,192.168.2.100,127.0.0.1,kubemaster,kubernetes.default
-profile=kubernetes kubernetes-csr.json | cfssljson -bare kubernetes
```

The following Google Docs "suggestion" is someone else's config that I'm leaving here to be nice. It does not match the IPs I used in this exercise, but illustrates how these things can change based on your personal configuration:

```
cfssl gencert  -ca=ca.pem   -ca-key=ca-key.pem  -config=ca-config.json
-hostname=10.2.0.15,192.168.56.80,127.0.0.1,kubemaster,kubernetes.default
-profile=kubernetes kubernetes-csr.json | cfssljson -bare kubernetes
```

This lists the hostnames of kubemaster (the controller's actual name), and also kubernetes.default, which I will add to the 3 nodes' hosts files to also match the IP of kubemaster.

Now, the etcd Certificate. (*trumpet fanfare*)

```
$ cat ketcd-csr.json
{
  "CN": "ketcd",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "Kubernetes",
      "OU": "Kubernetes The Hard Way",
      "ST": "Oregon"
    }
  ]
}
```

```
$ cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.json
-hostname=192.168.2.102,127.0.0.1,ketcd -profile=kubernetes ketcd-csr.json | cfssljson -bare
ketcd
```

Okay, assuming this works... what's next? Oh, yeah, distributing the keys with ssh, from the kubemaster.

```
$ cp ca.pem ca-key.pem kubernetes-key.pem kubernetes.pem ~
$ scp ca.pem kubenode-key.pem kubenode.pem kubenode:~/
$ scp ca.pem ketcd-key.pem ketcd.pem ketcd:~/
```

Generating Kubernetes Configuration Files for Authentication (5)

Note: $Kubernetes_Public_Address is the kubemaster or the Load Balancer/Ingress, if you have one. If you decided to access the whole thing through the Untangle box, this would be the public IP of that box. You could set the variable directly to make it easier (example: $ KUBERNETES_PUBLIC_ADDRESS=192.168.2.100), or you could type it in each time.

The configs must be generated from the same folder that the .pem's exist in (because of the --embed-certs=true).

Generating the Data Encryption Config and Key (6)

No changes.

Bootstrapping the etcd Cluster (7)

This is where we start to diverge, as I used an external etcd node. I did this explicitly for practice with etcd management. The config of 1 vCPU and 2GB RAM should support this tiny exercise environment, but not much more. You definitely wouldn't run any serious load against it, especially with a shared laptop CPU.

First, remember we made a separate key for etcd, so the filenames are changed:

```
$ sudo cp ca.pem ketcd-key.pem ketcd.pem /etc/etcd/
```

The etcd.service file must be edited to reflect the controller information in the --initial-cluster section, instead of using the exercise's defaults. You'll also need to replace the $INTERNAL_IP with the actual IP of the VM. Also also, the key names are different.

Other than all that, the only difference is a single node etcd cluster, that is NOT on the k8s controller node (and uses its own certs).

I recommend reading the docs before committing the service definition:
(https://coreos.com/etcd/docs/latest/v2/configuration.html)

```
cat > etcd.service <<EOF
[Unit]
Description=etcd
Documentation=https://github.com/coreos

[Service]
ExecStart=/usr/local/bin/etcd \\
  --name ketcd \\
  --cert-file=/etc/etcd/ketcd.pem \\
  --key-file=/etc/etcd/ketcd-key.pem \\
  --peer-cert-file=/etc/etcd/ketcd.pem \\
  --peer-key-file=/etc/etcd/ketcd-key.pem \\
  --trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-client-cert-auth \\
  --client-cert-auth \\
  --initial-advertise-peer-urls https://192.168.2.102:2380 \\
  --listen-peer-urls https://192.168.2.102:2380 \\
  --listen-client-urls https://192.168.2.102:2379,http://127.0.0.1:2379 \\
  --advertise-client-urls https://192.168.2.102:2379 \\
  --initial-cluster-token etcd-cluster-0 \\
  --initial-cluster ketcd=https://192.168.2.102:2380 \\
  --initial-cluster-state new \\
  --data-dir=/var/lib/etcd
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF
```

Bootstrapping the Kubernetes Control Plane (8)

We need to add the etcd cert flags to the .service file

```
$ sudo cp ketcd-key.pem /var/lib/kubernetes
$ sudo cp ketcd.pem /var/lib/kubernetes
```

API Server Service changes:

```
  --advertise-address=192.168.2.100 \\
  --apiserver-count=1 \\
  --etcd-certfile=/var/lib/kubernetes/ketcd.pem \\
  --etcd-keyfile=/var/lib/kubernetes/ketcd-key.pem \\
  --etcd-servers=https://192.168.2.102:2379 \\
```

Kubernetes-controller-manager AND kube-scheduler services changes:

```
 --master=http://192.168.2.100:8080 \\
```

We’re successful up to this point.

Okay, this lab exercise changes the configs not inconsiderately.

When creating the kube-apiserver.service file:
Replace $INTERNAL_IP with the IP of the kubeadm server

--apiserver-count=1, not 3 (unless you built 3)
--etcd-servers only list the IP of the single etcd node (keeping the port 2379)
--service-cluster-ip-range : this range is something you'll use later, just define it to something not currently in use. The range from Kelsey's docs worked fine for me, but it could be anything, as long as you're consistent.

Under the kube-controller-manager.service:
--cluster-cidr: This one's interesting. The IP range is for the pods, but if there's a firewall in front of the installation, the pod IPs need to be added to the definitions, as during the step Firewall Rules, during exercise 3. I kept the IPs unchanged, because I was on a Host-Only network.
--master, change $INTERNAL_IP to the kubeadm node's IP
--service-cluster-ip-range must match whatever you put above…
In the end, I only updated the internal IP.

Under kube-scheduler.service:
Just update the $INTERNAL_IP with your kubeadm node's IP

The Frontend Load Balancer:
I SKIPPED THIS ENTIRE SECTION because I have single nodes. I may come back and detail creation of an HAProxy instance to backfill this exercise. But not today...

If the "Verification" step fails, it's probably a certificate error. Okay, it's almost definitely a certificate error. I spent a whole day debugging errors here before I found I left my etcd node's IP out of the kubernetes.pem certificate. I solved this by having separate certs for etcd.

Bootstrapping the Kubernetes Worker Nodes (9)

First of all, this particular section gave me literally weeks of headache. In the end, it was a very simple fix, but the debugging was extensive.

Here (in Red) are all the ways (I remember/have documented) that I've taken tangents while trying to figure out what was wrong with this system. You can safely skip them, as they are just here for posterity and information:

- Investigate --flags in the service definition (ended up being the correct method for fixing this, in the end)
    - Removed CRI-O and CNI, due to errors about "remote" runtime
    - In at least one version --require-kubeconfig errored as "invalid parameter" when I stepped through running the kubelet command manually
    - In the end, it was a missing flag, --api-servers, which was labeled as deprecated and even removed from the 1.7 kubelet documentation, that was required because it wasn't being properly read from the kubeconfig
- Investigate Key Creation

    - Rebuilt keys with permutations of CN’s
- Investigate CRI-O
    - Investigation of whether or not I still needed Docker, at least to pull images.
    - Tried compiling from source
        - Turns out the latest CRI-O uses a function in go 1.8, so compiling with go 1.7 does NOT work.
    - Note: ocid and ocic were replaced by crio and crioctl, respectively.
- Ensuring Network Communication
    - $ sudo tcpdump -i enp0s3 -vvv | grep 192.168.2.101 (did not work)
    - $ sudo tcpdump -i enp0s3 -vvv | grep 6443 (DID work, because of /etc/hosts)
    - All the network traffic stopped, when I stopped the kube-proxy, which told me the kubelet was NOT performing any communication at all.
    - Checking to see which parts of kubernetes were seen with and without kubelet registration:
        - for i in `kubectl get 2>&1 | egrep '(\*)' | awk '{ print $2 }'`; do echo "kubectl get $i" ; kubectl get $i ; echo "-----" ; done
- Running the Kubelet manually instead of as a Service
    - This allowed me to more easily parse through the errors I got back, but ultimately, I still missed the one vital line that told me my error
        - W1113 14:51:54.207457   3032 server.go:496] **No API client: no api servers specified**
- Digging into etcd, to see if there’s comm problems
    - The API server was talking to etcd, but never registered any of the nodes
    - $ ETCDCTL_API=3 etcdctl get --from-key '' --write-out=fields | grep -i key | grep -vi value
        - That’s --from-key followed by 2 single quotes
- Started digging through the actual code
    - Git clone https://github.com/kubernetes/kubernetes.git
    - Dug through the kube-proxy code while trying to figure out the kube-proxy communication
    - Dug through the kubelet code looking for where it registered with the kube-api
    - None of it helped, compared to the other errors that I read 375 times, and acknowledged on the 376th time.
- Investigate cAdvisor errors about rkt
    - curl http://192.168.2.101:4194/validate/ to check cAdvisor’s state
        - Errors led to a rabbit hole about garbage collection that, in the end, was a complete red herring
    - This is a cAdvisor reporting error and not relevant.
        - W1113 14:43:02.479664   2673 manager.go:151] unable to connect to Rkt api service: rkt: cannot tcp Dial rkt api service: dial tcp [::1]:15441: getsockopt: connection refused

-----

For 1.7.4, you will need to add --api-servers=https://192.168.2.100:6443 (or whatever your controller's IP is) to the kubelet.service definition. This is not listed in the kubelet documentation, as it is supposed to be deprecated use, with the functionality pushed into the kubeconfig. Turns out that doesn't work as expected.

Now then... Once I got the --api-servers flag in place, I didn't need docker, or new certs, or really anything else. This section mostly went as written. It also correctly added the /registry/minions/kubenode to etcd, so it registered and communicated! Finding the solution revolved around https://github.com/kubernetes/kubernetes/issues/36745

The download and installation remains mostly unchanged.

In the bridge.conf, POD_CIDR would have been defined (and probably overlooked) during the gcloud commands creating the workers in exercise 3. We can technically use any unused range we have available, and since these are all on a host-only network, we can just pick a range. Since we only have one node, I'm just picking the first listed range from back in exercise 3, using 10.200.0.0/24. Note that this is a subnet (a /24 of a /16, in my case) of the CLUSTER_CIDR defined in the kube-controller-manager.service on the controller. Make sure they line up.

In the kubelet.service definition, we again need to replace the POD_CIDR, as above.

<u>Configuring kubectl for Remote Access (10)</u>

Just to test, I'm going to move the admin certs over to my local machine and run some kubectl commands from there, but I fully intend to run all the kubectl commands directly on the controller node, so this step is mostly irrelevant.

Yup, this worked fine, using the $KUBERNETS_PUBLIC_ADDRESS of 192.168.2.100

<u>Provisioning Pod Network Routes (11)</u>

The purpose of this exercise is irrelevant in a configuration with a single worker node. We may have to set up an explicit route for the pods to go to the worker node… This would be done manually with "route add" inside the Linux system, not handled outside it, as gcloud would do. Note to self: Figure out what that route add would look like.

<u>Deploying the DNS Cluster Add-on (12)</u>

Let's see if this works without docker to pull the images for CRI-O… It does!

This section simply worked as written, which it SHOULD, since the whole point of abstracting apps to a container-based system is that once you have a running one, it should run anywhere.

<u>Smoke Test (13)</u>

If you want to run the first check, just run the “ETCDCTL_API=3 etcdctl get /registry/secrets/default/kubernetes-the-hard-way | hexdump -C” command directly on the etcd host.

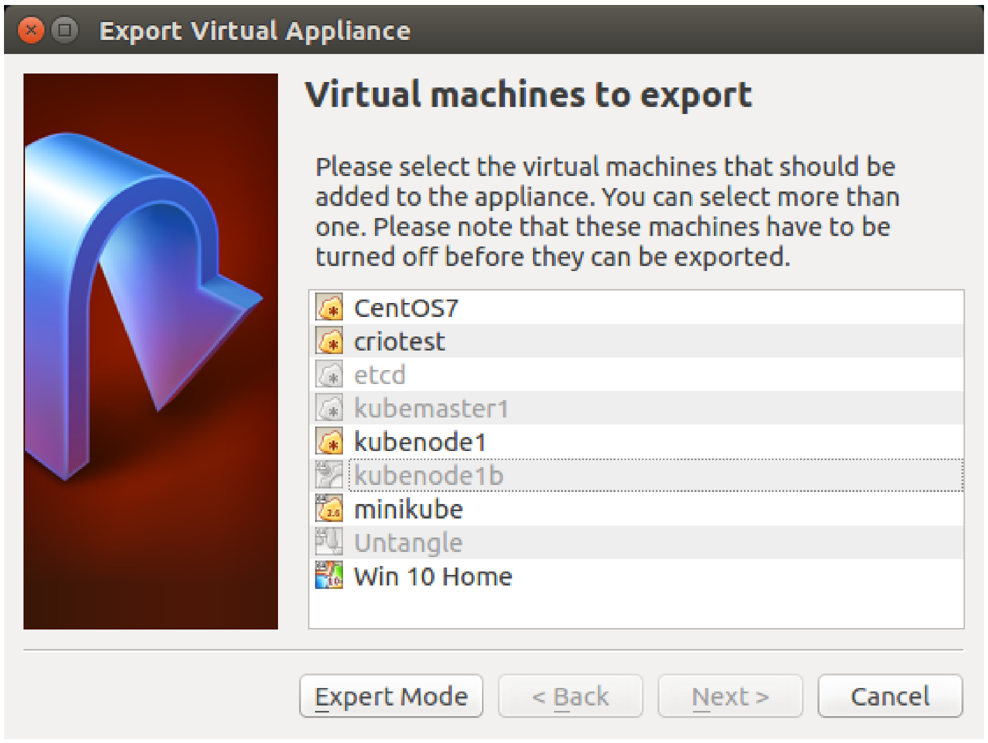
I stand corrected. The port forwarding exercise here will not run from the kube master, so you cannot run everything on the kube master, as I suggested in step 10. You will HAVE to run that one from an external system, such as the laptop I am currently running the cluster VMs on.

In the Services section, there is no external IP, so you’ll have to just use the IP of the kubenode (192.168.2.101, in my case).

You can ignore the firewall rules, unless you REALLY want to go set up static routing and NAT inside the Untangle box, just to get to a test service. I suggest you don’t bother.

<u>Cleaning Up (14)</u>

Delete all the VMs and start over. Actually, don’t clean up. Instead, shutdown all 4 of the VMs, then export them into OVAs (assuming you have about 5 free GB of disk space), so you can restore these to this state and have a backup of a fully built cluster.


Export Virtual Appliance

Virtual machines to export

Please select the virtual machines that should be added to the appliance. You can select more than one. Please note that these machines have to be turned off before they can be exported.

CentOS7
criotest
etcd
kubemaster1
kubenode1
kubenode1b
minikube
Untangle
Win 10 Home

We’ll talk about VBox image archiving, so you can break the installation with new network plugins or such, and then restore. Now that you have working images, you can do a lot with them.